Динамично програмиране

Тодор Дуков

Какво е динамично програмиране?

Динамичното програмиране не е нито динамично, нито програмиране. Това е както оптимизационен метод, така и алгоритмична парадигма, която е разработена от Ричард Белман през 50-те години на миналия век. В този метод една зачача се разделя на подзадачи по рекурсивен начин. Той се използва в два случая:

- при задачи, които имат припокриваща се подструктура т.е. задачата се разделя на подзадачи, които се срещат няколко пъти
- при задачи, които имат оптимална подструктура т.е. оптимално решение може да се конструира от оптимални решения на подзадачите

Прости примери за динамично програмиране

Да кажем, че искаме да сметнем n-тото число на Фибоначи. Един начин е да караме по рекурентното уравнение:

```
int fibonacci_recursive(int n)
{
    if (n < 2)
        return n;

    return fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2);
}
```

Проблемът е, че получаваме експоненциална сложност по време. Нещо, което можем да направим, е да пазим вече пресметнатите стойности, за да не се налага да ги пресмятаме пак:

```
int fibonacci_dp(int n)
{
    if (n < 2)
        return n;

    int dp[n + 1];
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; ++i)
    {
        dp[i] = dp[i - 1] + dp[i - 2];
    }

    return dp[n];
}
```

Това е пример за задача с припокриваща се подструктура, с решение по схемата **динамично програмиране**. Успяхме да решим задачата за линейно време.

Нека сега видим пример за задача с оптимална подструктура. Да кажем, че имаме два низа $S_1[1 \dots n]$ и $S_2[1 \dots m]$ и искаме да пресметнем дължината на най-дългата обща подредица на S_1 и S_2. Лесно се вижда, че дължината $\mathrm{LCS}_{S_1,S_2}(i,j)$ на най-дългата подредица на $S_1[1 \dots i]$ и $S_2[1 \dots j]$ може да се пресметне рекурсивно така:

$$\mathrm{LCS}_{S_1,S_2}(i,j) = \begin{cases} 0 & \text{, ако } i = 0 \text{ или } j = 0 \\ \mathrm{LCS}_{S_1,S_2}(i-1,j-1)+1 & \text{, ако } i,j > 0 \text{ и } S_1[i] = S_2[j] \\ \max\{\mathrm{LCS}_{S_1,S_2}(i-1,j), \mathrm{LCS}_{S_1,S_2}(i,j-1)\} & \text{, ако } i,j > 0 \text{ и } S_1[i] \neq S_2[j] \end{cases}$$

Ако искаме да пресметнем това със обикновена рекурсия, отново ще получим експоненциална сложност по време.

Отново можем да направим решение по схемата **динамично програмиране** със сложност $\Theta(n \cdot m)$:

```
int longest_common_subsequence(char *s1, int n, char *s2, int m)
{
    int dp[n + 1][m + 1];

    for (int i = 0; i <= n; ++i)
    {
        dp[i][0] = 0;
    }

    for (int j = 0; j <= m; ++j)
    {
        dp[0][j] = 0;
    }

    for (int i = 1; i <= n; ++i)
    {
        for (int j = 1; j <= m; ++j)
        {
            if (s1[i - 1] == s2[j - 1])
                dp[i][j] = dp[i - 1][j - 1] + 1;
            else
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
        }
    }

    return dp[n][m];
}
```

Динамично програмиране за решаване на комбинаторни задачи

Вижда се, че тази техника е много удобна за бързо пресмятане на рекурентни зависимости. Едно приложение е в решаването на комбинаторни задачи. Да кажем, че искаме да пресметнем бързо $\binom{n}{k}$. Нека първо си припомним дефиницията:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Единственото нещо, което се иска да сметнем два факториела:

```
int binomial_factorial(int n, int k)
{
    if (n < k || n < 0 || k < 0)
        return 0;

    int fact[n + 1];
    fact[0] = 1;

    for (int i = 1; i <= n; ++i)
    {
        fact[i] *= fact[i - 1];
    }

    return fact[n] / (fact[k] * fact[n - k]);
}
```

Получихме сложност по време $\Theta(n)$, вместо $\Theta(n \cdot k \cdot (n-k))$. Обаче решението не е практично. Проблемът е, че функцията $n!$ расте много бързо. Ние ще работим с големи стойности във `fact` масива, но крайният отговор ще е много по-малък от това. Ако искаме да си гарантираме възможно най-малки стойности по време на изчисление, трябва да го пресметнем за време $\Theta(n \cdot k)$ чрез формулата:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Нека сега се опитаме да пресметнем броят T_n[*] на двоични дървета за търсене с n различни върха. При $n = 0$ положението е ясно. Ако $n \geq 1$, то имаме няколко случая в зависимост от това кой връх е корен на дървото. Броят на двоичните дървета за търсене с корен i-тия по големина връх, където $1 \leq i \leq n$, е равен на броя двоичните дървета с $(i-1)$-те по-малки от него върха, умножен по броя на двоичните дървета с останалите $n-i$ върха. Това е точно $T_{i-1} \cdot T_{n-i}$. Така получаваме следното рекурентно уравнение:

$$T_0 = 1$$

$$T_n = \sum_{i=1}^{n} T_{i-1} \cdot T_{n-i} \text{ за } n > 0$$

Ясно е, че не искаме да пресмятаме T_n чрез рекурсия – това би било кошмарно бавно. Отново ще помним предишни изчисления, за да си забързаме алгоритъма до такъв със сложност $\Theta(n^2)$:

```
int catalan(int n)
{
    int dp[n + 1];
    dp[0] = 1;

    for (int i = 1; i <= n; ++i)
    {
        dp[i] = 0;
        for (int j = 1; j <= i; ++j)
        {
            dp[i] += dp[j - 1] * dp[i - j];
        }
    }

    return dp[n];
}
```

Задачи

Задача 1. Да се напише колкото се може по-бърз алгоритъм, който пресмята $\binom{n}{k}$ с рекурентната формула от по-горе. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 2. Да се напише колкото се може по-бърз алгоритъм, който при подадено естествено число n, намира броят на начини човек да се изкачи по тях, като може да изкачва най-много 3 стъпала наведнъж. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 3. Да се напише колкото се може по-бърз алгоритъм, който при подадена булева матрица $A[1 \ldots n, 1 \ldots m]$ намира броя на пътищата (движейки се само надясно и надолу) от $(1,1)$ до (n,m), които не минават през 1 в A. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 4. Да се напише колкото се може по-бърз алгоритъм, който при подадена матрица от естествени числа $A[1 \ldots n, 1 \ldots m]$ намира минималната цена на път (движейки се само надясно и надолу) от $(1,1)$ до (n,m), като под цена на път разбираме сумата на всичките $A[i,j]$, срещнати по пътя. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 5. Да се напише колкото се може по-бърз алгоритъм, който при подаден целочислен масив $A[1 \ldots n]$ намира дължината на най-дългата строго растяща негова подредица. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 6. Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от естествени числа $A[1 \ldots n]$ и естествено число s намира броя на начините, по които могат да се изберат елементи на A, така че да сумата им да е s. След това да се докаже неговата коректност, и да се изследва сложността му по време.

Задача 7. За масив от положителни числа $A[1 \ldots n]$ и $1 \leq i < j \leq n$ казваме, че можем да стигнем от i до j в A за една стъпка, ако и $j - i \leq A[i]$. Да се напише колкото се може по-бърз алгоритъм, който при подаден масив от положителни числа $A[1 \ldots n]$ намира минималния брой стъпки, с който можем да стигнем от 1 до n в A. След това да се докаже неговата коректност, и да се изследва сложността му по време.

[*] Тези числа се наричат числа на Каталан.